Definitive programming for parallelism

(draft paper)

Meurig Beynon

Department of Computer Science
University of Warwick
Coventry CV4 7AL

Area of interest

Concepts and paradigms for concurrent systems / Models for parallelism

Abstract

Programming language principles are of fundamental importance in supporting major applications on multiprocessor architectures. Many different language paradigms for general-purpose parallel computing have been proposed. The most appropriate choice of paradigm remains contentious and problematical.

A novel approach to concurrent programming, based upon the formulation of definitions ("definitive programming"), is outlined. A case for the further investigation and development of the definitive programming paradigm as a basis for general-purpose parallel computing is presented. The paper builds upon previous work by the author on the application of definitive principles to the implementation of interactive systems, and the modelling and simulation of concurrent systems. The general principles and specific features of definitive programming are discussed in the context of existing programming paradigms, and the relative merits of definitive programming for general-purpose parallelism are considered. An appropriate abstract machine model is motivated, described and illustrated with reference to the simulation of a simple concurrent system.

Introduction

In the search for methods of exploiting massive parallelism, the importance of a coherent approach to the development of architectures, languages and applications is well-recognised [16]. It may be argued that the issue with the most significant implications for all three aspects of a parallel computing project is the choice of the underlying programming paradigm. As the diverse programming styles under investigation within the Parallel Architectures and Languages Europe project illustrate [4,5,14,16], the most appropriate choice - if such exists - remains controversial. Indeed, it has been argued that multiprocessors cannot be programmed effectively without radical developments in programming langage design [2].

The purpose of this paper is to propose an approach to parallel computation based upon a novel programming paradigm - "definitive" (for "definition-based") programming. The development of this paradigm has been the focus of research carried out at the University of Warwick under the author's direction over several years [6,7,8,9,10,11]. At this stage, it is impossible to deal comprehensively and formally with the many issues raised by this research programme. The intention is rather to report upon work in progress, and to indicate some of the reasons, both abstract and technical, why there may be good prospects for parallelism.

In evaluating a programming paradigm for parallelism, there are several key questions to address. In the spirit of [2], we may examine the way in which features of a language support or obstruct various kinds of parallel action. Recalling the need to integrate a parallel programming language with both architecture and applications, we must also consider what kind of abstract machine model should be adopted, and how effectively such a computational model can be applied. A typical approach to research into general-purpose parallel programming has been to separate the language issues into two categories: high-level language issues, concerned with the applications interface, and operating system issues, concerned with the architecture interface. The methods presently most favoured for high-level specification of software are declarative in nature, whilst scheduling processors has traditionally relied upon procedural techniques to address issues of timing and synchronisation. This makes it difficult to maintain a consistent programming paradigm at all the different levels of abstraction that lie between the application programmer and the parallel machine hardware. Unfortunately, any inconsistency is potentially an unwelcome source of complexity and inefficiency. Such issues have had to be addressed in the Alvey Flagship project, for instance [3].

This paper includes an informal overview of the principles of definitive programming. A central concept of definitive programming is the reconciliation of declarative and procedural computational views [6]. Other potential advantages where parallelisation is concerned are discussed with particular reference to a recent critique of existing language paradigms by Baldwin [2]. An abstract machine model for definitive programming is motivated and informally described. The use of this model to support the simulation of a simple concurrent system is illustrated. The significance of the abstract machine model within the overall framework of research into programming with definitive principles is briefly considered.

Definitive programming: an overview

Definitive programming is based upon the use of definitions that establish functional relationships between variable values. The primary concepts originated from using

definitions as a means of describing an interaction between the user and the computer [6] - a principle illustrated in its simplest form by a spreadsheet stripped of its tabular interface. Subsequent developments have led to the elaboration of a general-purpose programming paradigm that is associated with an abstract machine model to be described below. In this paper, some familiarity with the principles of this style of programming, such as is set out in [6] and [9], will be assumed, and the essential concepts will be informally introduced. Only those issues most relevant to parallelisation will be considered. For details of what is involved in using definitions to describe complex data types - for instance - the interested reader may refer to other papers [6,9,11].

In definitive programming, the term *definition* is used in a technical sense. Formally, a definition is program statement of the form

variable = *formula*

in which the *formula* is a recipe (expressed in terms of algebraic operations to be performed on the values of other variables and constants) for determining the value of the *variable*. Such a definition is to be understood as asserting that the value of the *variable* is determined by the *formula*, so that changes to the values of variables appearing in the *formula* will in general affect the value of the *variable*. In this way, a system of definitions establishes a set of dependencies between variable values that - to be meaningful - must at all times be free of cycles of dependency relations. As a simple example, the system of definitions:

```
resistance = resistance_of_lamp + length * coefficient_of_resistance(1.1)
current = if switch_on then voltage / resistance else 0
light_on = switch_on and current >= min
switch_on = false
```

may be used to represent the state of an electrical circuit comprising a power source, a switch, a light and a variable resistance such as is supplied by a simple rheostat. Several characteristic features of such a definitive system may be noted. The order in which definitions are interpreted is unimportant. There will generally be variables whose value is specified explicitly but is subject to change, such as "length" and "switch_on" above, constants such as "resistance_of_lamp" and "min" (the minimum current required to light the lamp) and dependent variables (such as "light_on"). It is also possible for the value of a variable (such as "voltage") to be unspecified.

A definition superficially resembles a constraint, but differs in two significant respects. In the first place, the constraint it establishes is uni-directional. In the second place, a definition implicitly incorporates a particular - albeit specific and limited - method for maintaining a constraint viz "update the value of the variable on the left when the value of a variable that appears in the formula on the right changes". The "definitions" introduced as an auxiliary feature in a functional programming environment are in some respects very similar, but are used in an entirely different way. A definition such as appears in a MIRANDA script, for instance, gives a name to particular function or result of a function evaluation, but - to preserve referential transparency - re-assignment to such a name is not permitted [18]. It is of course possible to simulate the effect of such re-assignment in the MIRANDA environment (viz by editing a script), but this is outside the scope of the functional programming idiom. There is also a similarity between the use of definitions in a definitive environment and that of macros in a procedural context. The correspondence between the concepts is very precise, but in practice their role is dissimilar. For instance, it would be most unusual for a procedural program to dynamically redefine a macro definition. (For an account of the programming potential of macros, see Strachey [17].)

To understand the significance of using definitions within a framework that forfeits referential transparency, it is most instructive to compare programming in a definitive style with typical purely procedural programming. A common pattern to be observed in procedural programming is the execution of a sequence of linked assignments, such as

{x==2} x:=3; b:=a+c; d:=b+x+e; f:= g[b]

(1.2)

in which - after the first assignment, that changes the value of x - it becomes necessary to update the values of several other variables. From one perspective, the sequence of assignments (1.2) may be seen as ensuring the preservation of an appropriate invariant. What is particularly significant is that the order of execution of assignments is generally crucial - a matter of great concern for parallelisation. In a definitive programming framework, the role of definitions is to express the dependencies among variables values as they relate to any proposed procedural action (in this context: variable re-definition). On the assumption that the assignments in (1.2) represent all the updates that are consequences of assigning a new value to x, the formulation within a definitive programming environment might become

{x==2} b=x+c; d=b+x+e; f=g[b]; x=3;

(1.3)

in which the context for the re-assignment x=3 is established by the prior definitions of b, d and f. Innocuous as this change of viewpoint at first appears, there are some important consequences. In the first place the order in which definitions are introduced is much less tightly constrained. What is more, if it is subsequently necessary to change the value of x a second time, the sequence of assignments in (1.2) must be re-executed, whilst in (1.3) a simple reassignment to x will suffice. It may also be noted that the approach adopted in (1.3) alleviates some of the traditional problems of processing in a Von Neumann model - introducing extra storage of relations in order to reduce the amount of communication of values between the processor and the memory.

The prescriptive implications of the above ideas will be explored more fully below. By way of background, there are some interesting general issues to be addressed, with reference both to other programming paradigms, and to parallelism in particular. It may be seen that definitive programming is neither purely procedural nor purely declarative. The validity of its computational model rests upon the premise that the effect of any action of an agent engaged in a computation can be effectively represented using the paradigm illustrated in (1.3). In some sense, a system of definitions declares the context for an action that is subsequently performed through a redefinition (or family of redefinitions). In a particular application, the synthesis of declarative and procedural principles that definitive programming entails is to this extent prescribed by semantic considerations. This may be contrasted with an adherence to principles - such as "referential transparency" in declarative programming, or "information hiding" in object-oriented programming - that are adopted for methodological or technical reasons. (The Preface to [13] makes a similar contention that "functional programming is more problem-oriented than conventional languages". This claim is most convincing for problems in which referential transparency is easily attained.)

In [2], Baldwin explores several key issues that determine how suitable a particular programming paradigm is for parallelisation. He identifies

"two deep flaws of existing languages:

1) reliance on side-effects,

2) use of iteration or recursion to express data parallelism ...".

He also describes "the ultimate goal for a parallel programming language" as "supporting a clear statement of the data dependencies". As the above discussion illustrates, the application of definitive principles is centrally concerned with the rationalisation of side-effects. As will become clear from the design of the abstract machine model below, definitive methods have such power to express computational state that complex procedural abstractions are unnecessary. What is more, a system of definitions can express the dependencies between variable values in a manner that is both explicit and independent of ordering. The potential for interference in parallel procedural programming typically stems from consideration of such issues as

"has this variable currently an appropriate value?";

and is clearly a critical problem in the context of sequences of assignments such as (1.2). In functional languages, interference typically arises from such concerns as:

"is this variable currently defined?";

a problem that can be alleviated within a definitive framework, since undefined values can be gracefully accommodated. (It may even be possible to process systems of definitions that cannot yet be evaluated in their symbolic form so as to expedite the computation of values when these first become defined.) All these observations - motivated by Baldwin's analysis - suggest that definitive programming has clear potential for parallelisation. Superficially, it may also appear that advocacy of definitive programming is in spirit consistent with Baldwin's personal view that constraint-based programming approaches offer the best prospects for general-purpose parallel programming [2]. In fact, a closer examination of the issues identifies significant differences with important implications for parallelisation.

A definitive approach to concurrent programming

To motivate the abstract machine model for definitive programming to be described below, it is helpful to look more critically at the computational abstraction underlying the use of definitions. Note first that - because of its procedural ingredients - definitive programming cannot readily be accommodated within a purely declarative framework. On the contrary, definitive programming reverts to the traditional view of a computation as a sequence of transitions from state to state. Naively - adopting the perspective of [6], for instance - it may seem that computational states could be effectively represented by systems of definitions alone. A more precise analysis of the programming paradigm illustrated in (1.3) above reveals that a single system of definitions, rather than representing the computational state in a comprehensive manner, "supplies the context for one or more potential actions" (c.f. the discussion of "intelligent views" in [9]).

To develop this idea, recall that the definitions of b, d and f in (1.3), once established, make it possible to redefine the variable x any number of times. In a program of practical interest, we should not necessarily expect such simple patterns of re-assignment to be fixed for all time. (This is of course convenient for a variety of problems if the relationships in (1.3) are sufficiently complex - when it becomes a paradigm not unlike pure functional programming. That is, it resembles writing a set of function definitions, and performing a sequence of function evaluations, though in functional programming these evaluations can have no side-effects on the computational state.) The necessity for changing definitive systems governing particular sets of variables according to the context often has a direct interpretation in respect of models developed for a particular application. The definitive system (1.1) is a valid model of the way in which the status of a circuit varies as the switch is reset, and the rheostat is adjusted. It ceases to be valid if the light bulb fails. It would not be as faithful a representation of an electrical circuit in which an electric motor was substituted for the

light, since the resistance of such a motor would vary according to the current load. And if this motor were an alternator that served both as a starter motor and a dynamo, the entire format of the definitive system would have to be radically altered.

The above examples illustrate both how a definitive system must initially be chosen to reflect the semantics of the application, and how it must be modified in general as a computation or simulation progresses, to reflect the changing context within which a procedural action operates. In simple terms, the implications of re-assigning the variable x, as expressed through a definitive system such as is given in (1.3), may be context-sensitive. Of equal relevance is the need in general to change the context if some other action is to be executed. As a simple example, were the variable b in (1.3) to be reassigned so that the relationships expressed in (1.3) remained invariant, the appropriate context would be established by eliminating the definition "b=x+c" in favour of "x=c-b". When more than one agent is involved, as in a concurrent system, there may be no single definitive system consistent with potential actions: such a situation constitutes interference.

The above discussion supplies the fundamental concepts behind the abstract machine model to be more formally described below. It also puts into a clearer perspective the relationship between defintive programming and other paradigms, and suggests other potential advantages.

In as much as it enables several simple procedural actions - as in (1.2) - to be viewed as a single complex action - as in (1.3), definitive programming is akin to object-oriented programming. The relationships between variable values specified by definitions may resemble those established by message passing between objects, and the operators that appear on the RHS of definitions are associated with evaluations (resembling invocations of methods) that are hidden and implicitly specified. As the work of America on the semantics of POOL illustrates [1], one of the problems in concurrent object-oriented environments is to clarify precisely and abstractly what relationships are established, taking account of the complex ways in which messages and the execution of methods may be synchronised. The representation of relationships supported by definitive systems has patent advantages in these respects. One noteworthy distinction is that in the abstract definitive machine below there is no concern about "information hiding": one entity can act directly to change the values of variables bound to another.

As we have remarked above, definitive programming is set apart from functional programming through its non-declarative aspects. It may at first seem reasonable to regard definitions as predicates about the state of a model, resembling the predicates that appear in a logic program or the constraints in a constraint-based environment. This is to overlook the fact that the significance of definitions is closely linked with action and "change of state", rather than with static inference from logical assumptions about a particular state. To this extent, definitive programming has more affinity with procedural programming with invariants, in its sequential and concurrent variants [12,15].

The relationship between definitive programming and procedural programming with invariants deserves further elaboration. It is customary to model a complex process involving several agents in terms of "performing certain actions and achieving certain states". In the process of buying a house, for instance, actions might be: "Putting the old house up for sale" "Paying a deposit" "Arranging for a survey" "Negotiating a mortgage". States might be represented by boolean combinations of predicates such

as: "I have put my house on the market" "I have signed the contract" "I have paid the deposit" "we have exchanged contracts". The predicates required to model the intermediate states in a process are characteristic of the process; when suitably combined they can express pre-conditions and post-conditions of actions within the process. Before I sign the contract, I must negotiate a mortgage; when we have exchanged contracts I have sold my house. Within the definitive programming framework proposed below, the logical relationships that provide the characteristic framework for the actions of a process must themselves be modelled by definitive systems in which the variables designate appropriate boolean conditions.

Of course the same process that allows us to interpret "buying a new house" as a sequence of actions to effect the transition through particular states can be applied to the individual constituent actions within the house-buying process. Thus "signing the contract" may involve "visiting the solicitor" "receiving a document from the solicitor" "finding a pen" "writing my name on the document" "returning the document to the solicitor" etc etc. In refining a specification in this manner, there comes a point beyond which further elaboration of actions is difficult, and a concept of atomic action is encountered. At such a low-level of abstraction we should like to be able to characterise the actions by pre- and post- condition states that differ "by a very small amount". In this process of refinement, an important principle emerges. The significance of a low-level action is very context dependent (I am signing a form vs I am doodling on the telephone pad), and may be "precisely synchronised" with a significant action ("I am accepting liability") at a higher level of abstraction. A celebrated quotation illustrating this principle is:

"A small step for a man, a giant step for mankind".

As (1.3) indicates, a definitive programming paradigm can capture the context-sensitivity of low-level action, and provide a clear formulation of how such an action is synchronised with other significant changes of state.

The context-sensitivity of actions is a major obstacle when identifying interference. In an appropriate context, a little action can have great consequences, and its potential for interference may be disproportionate. As (1.3) suggests, the use of a definitive paradigm reflects the full extent and influence of an action more faithfully than alternative paradigms. The possibility of using definitions to model processes in which a tiny action implicitly entails another - as already alluded to above -is one illustration of this. At the very moment that a cheque is passed by the bank, I buy an article, since by definition I own the article subject to several conditions, of which "having paid for the article" is generally the last to be satisfied. In some contexts, it might be appropriate to use a similar principle in a more liberal fashion. For example, the act of switching on an electrical appliance may be modelled in terms of its total effect (the switch clicks on, the light comes on, the motor starts, the wheels turn etc etc) rather as an isolated action, even though the synchronisation of action and side-effect is "in reality" approximate. The validity of such a model hinges upon whether the consequences of an action are deemed to be inevitable, or whether they can be averted through the intervention of another agent.

The abstract definitive machine as a model for concurrent action

The appropriate abstract machine model in which to formulate "definitive programming", as informally introduced above, may be seen as a generalisation of a Von Neumann architecture. The "abstract definitive machine" has a memory, and a processor. To each variable, there corresponds a memory location that can retain a definition - possibly implicit - rather than an explicit value. The machine code for the processor

is specified by a set of guarded actions to be executed in parallel as and when the guards allow. Each action is specified by a sequence of instructions. Each instruction either redefines a variable, or leads to the instantiation or deletion of an entity comprising a set of definitions and actions. A program takes the form of a set of abstractly specified entities. Execution is initiated by instantiating appropriate entities, and a computation terminates when there is no true guard.

As explained above, there are two constituents to procedural action in a definitive model. It is first necessary to establish the correct context for an action by introducing definitions, then to carry out an appropriate value re-assignment. Within the abstract definitive machine, there are two ways in which such activity can be supported. The definitions that establish a context can either be explicit within instantiated entities, or they can be introduced as a system of redefinitions within an action. For instance, the sequence of instructions (1.3) might be represented by an action of the form:

x_to_be_changed -> b=x+c; d=b+x+e; f=g[b]; x=3;

where the required context is dynamically established, or - if the appropriate context is already established - by the primitive action:

x_to_be_changed -> x=3.

Both kinds of procedural activity are illustrated in the example below.

The principal features of this "abstract definitive machine" model are depicted schematically in Figure 1 (for more details, see [11]). Execution follows a cyclic pattern, each cycle comprising the evaluation of guards in the context specified by the definitions currently stored in the definition store, and the parallel execution of those actions that are associated with true guards. Any evaluation of expressions required in a redefinition - as when "fixing the exchange rate" for purposes of a currency transaction - is performed in the same context as guard evaluation. Interference between actions can of course occur. For instance, the same variable may be redefined independently in concurrent actions, or the sequence of redefinitions may introduce cyclic dependency. Interference arises when the appropriate contexts for performing two or more actions cannot be realised concurrently. For the present, such problems are identified dynamically during program execution, though there may be some potential for static analysis.

The ideas behind the design of the abstract definitive machine are illustrated by the simulation of a simple concurrent system. Suppose that the blocks x and y are under the independent control of two agents. For simplicity, assume that the blocks are free to move in 1-dimension, have unit length, that their centres are always positioned at integral points px and py, and that they are always moved by steps of 1 unit in discrete actions. Assume also that x and y are connected by an inelastic string of integral length d>1. (This model makes most physical sense when the blocks are small, and d is very large.)

The outline specification in Figure 2 includes only the basic ingredients needed to describe the intended behaviour. (The annotations on the right are mnemonics that serve to identify and distinguish between actions.) The given skeleton must be complemented by adding a control() entity that provides the correct synchronisation between actions. The actions of the handler() entities for instance, must be sequential: since actions [^], [<] and [>] are simultaneously enabled, these must be made mutually exclusive. A simple method to ensure this is to generate an element from the set {< , ^ , >} at random within the control() entity, and to select the appropriate action accordingly. There are a number of more subtle omissions. The actions of the blockmover() entity interfere in several ways. A static analysis will establish that at most two ac-

tions of the blockmover() entity can be performed in each execution cycle: at most one from each of the sets {[<]~, [<]--, [>][?], [>]..} and {~[>], --[>], [?][<], ..[<]}. Certain combinations of action cannot arise: for instance, the preconditions for [<]-- and ~[<] are incompatible. Actions [<]-- and --[>] interfere on parallel execution: they correspond to a situation in which the string is taut and the handlers are pulling in opposite directions. There is a conflict between actions [<]-- and ..[<], in so far as concurrent action is only possible because actions specify movement through the same distance. The actions [>].. and ..[<] are in conflict when this entails a collision of the blocks at a single location.

The possible patterns of singular behaviour are summarised in Figure 2. Most of these will be dynamically detected as instances of interference. For instance, it is clear that the actions [<]-- and --[<] interfere, since they invoke an inconsistent system of definitions if executed in parallel. The conflict between the actions [>].. and ..[<] that arises specifically when py-px = 2 is not detected as interference since colocation of blocks is deemed impossible for reasons that relate to the semantics of blocks; another model might admit this possibility. The way in which the complementary condition (viz that associated with the actions [<]~ and ~[>] when py-px = d-1) is handled in the model illustrates one possible method for resolving exceptional behaviour. It is interesting to consider how different methods for dealing with the exceptional conditions can be interpreted. The conflict arising from the parallel execution of [<]-- and --[>] could be resolved by permitting no movement, by allowing one handler to dominate the other - whether arbitrarily or otherwise, or by deeming that the string snap.

The above discussion hints at how definitive principles can be used as the basis for a general-purpose programming language. In many applications, it should be possible to program directly using an enhanced variant of the abstract definitive machine in which more sophisticated data types and operators play the role of the integer and boolean variables in the above example. A fuller discussion of how the implementation of a CAD system can be approached along these lines is given in [9] (see also Figure 1). To address the issues raised by concurrency, whether at the level of application - as when modelling and simulating concurrent systems - or architecture - as when compiling for a distributed architecture, another perspective is required. (An analogous change of perspective distinguishes sequential and parallel object-oriented programming.)

Note that parallel activity within the abstract definitive machine is organised into concurrently instantiated entities, each capable of many synchronised parallel actions, rather than into concurrently acting sequential agents. When examining the true implications of concurrent activity, it is more appropriate to formulate a specification in terms of participating agents, and to use the abstract definitive machine model as a form of intermediate code. To support such an "agent-oriented" view, it is necessary to take account of the asynchronous behaviour of independent agents, and their need to cooperate through communication.

The basic issues can be illustrated with reference to the block moving example. Subject to ensuring that actions [<], [^] and [>] are mutually exclusive, the specifications of the handler() entities resemble sequentially acting agents. It is surely possible to prescribe realistic behaviour within the model by introducing an appropriate control() entity. The problem of specifying the protocols of the handler agents to achieve this pattern of control is quite another matter. For example, in what way must these agents respond in the context of a potential collision between blocks? To adequately address such concerns, it becomes necessary to model features of the current state to which

an agent can be responsive, and how this information can be communicated to other agents.

A full discussion of these issues is beyond the scope of this paper. The intention is that an appropriate specification can be expressed using the LSD notation [7,10] - an agent-oriented notation for representing concurrent systems that is also based upon definitive principles. Figure 2 superficially resembles an LSD specification, but the latter differs crucially in that actions are organised by agent, that the variables through which communication between agents is modelled must be specified, and that actions are executed asynchronously. The scope for complex synchronisation patterns within LSD models is perhaps comparable to that encountered in a parallel object-oriented environment, and there are many issues yet to be resolved [10]. The identification of the abstract definitive machine is seen as an important step towards giving a satisfactory formal account of the behavioural aspects of LSD models.

Conclusion

This paper seeks to promote the further investigation and development of definitive programming as a basis for general-purpose parallel programming. Definitive programming can be seen as integrating principles and concepts in existing language paradigms that have been studied in connection with parallel architectures. In many respects, it seems likely to meet the criteria required of an appropriate programming medium for multiprocessors, as identified by Baldwin in [2].

The abstract definitive machine model described appears to lie at an appropriate level of abstraction, potentially bridging the gap between application-oriented and architecture-oriented language concerns. Much research has already been directed at applying definitive principles to the implementation of interactive systems, and to modelling and simulation problems with concurrency. The initial indications are encouraging, but there are still many issues to be addressed, both in connection with applications, and where implementation on parallel architectures is concerned.

Acknowledgements

I am much indebted to Mark Norris of British Telecom Research Laboratories for promoting and encouraging my interest in applying definitive principles to concurrent systems. I am grateful to Mike Slade and Edward Yung for collaboration in designing and implementing the abstract definitive machine. I also wish to acknowledge the role that the financial support of British Telecom has indirectly played in stimulating this research - this should not be taken to imply any endorsement of the views expressed, for which the author accepts full responsibility.

References

1. P America *Object-oriented programming: a theoretician's introduction*, EATCS Bull #29, 1986
2. D Baldwin *Why we can't program multiprocessors the way we're trying to do it now*, Technical Report 224, Department of Computer Science, University of Rochester 1987
3. R Banach, P Watson *Dealing with state on Flagship: the MONSTR computational model*, Proc Conpar'88 (to appear)
4. B Bergsten, R Gonzalez-Rubio *A database accelerator and its languages*, Proc Conpar'88 (to appear)
5. G L Burn *Developing a distributed memory architecture for parallel graph reduction*, Proc Conpar'88 (to appear)
6. W M Beynon *Definitive notations for interaction*, Proc hci'85, CUP 1985
7. W M Beynon *The LSD notation for communicating systems*, CS RR#87, Warwick Univ, 1986
8. W M Beynon, Y W Yung, *Implementing a definitive notation for interactive graphics*, New Trends in Computer Graphics, ed Magenat-Thalman & Thalman, Springer-Verlag 1988, 456-68
9. W M Beynon, A J Cartwright *A definitive framework for implementing intelligent CAD systems*, in Proc 2nd Eurographics Workshop on Intelligent CAD Systems 1988 (to appear)
10. W M Beynon, M T Norris, M D Slade *Definitions for Modelling and Simulation of Concurrent Systems*, in Applied Simulation and Modelling, Proc IASTED ASM'88, Acta Press 1988, 94-98
11. W M Beynon, M D Slade, Y W Yung, *Parallel computation in definitive models*, Proc Conpar'88 (to appear)
12. K M Chandy, J Misra, *Parallel Program Design: a Foundation*, Addison-Wesley 1988
13. H Glaser, C Hankin, D Till *Principles of Functional Programming*, Prentice-Hall 1984
14. E Gluck-Hilltrop *The Stollman Data Flow Machine*, Proc Conpar'88 (to appear)
15. D Gries *The Science of Programming*, Springer-Verlag, 1981
16. E Odijk, W Bronnenberg *Parallel Computing: the object-oriented approach*, Proc Conpar'88 (to appear)
17. C Strachey *A general-purpose macrogenerator*, Computer Journal 8, 225-41, Oct 1965
18. *MIRANDA System Manual,* Research Software Ltd, 1987

Figure 2: An outline specification for the block moving simulation

```
entity handler(block)
{
    definition
        driving[block] = drivingL[block] or drivingR[block],
        drivingL[block] = holding[block] and pushingL[block],
        drivingR[block] = holding[block] and pushingR[block],
        pushingL[block] = false,
        pushingR[block] = false,
        holding[block] = false

    action
        not holding[block] -> holding[block] = true,
        holding[block] and not driving[block] -> holding[block] = false,
[^]
        holding[block] and not driving[block] -> pushingL[block] =
true,  [<]
        holding[block] and not driving[block] -> pushingR[block] =
true,  [>]
        drivingL[block] -> pushingL[block] = false,
        drivingR[block] -> pushingR[block] = false
}

entity blockstate()
{
    definition
        px, py, d,
        stringtaut = not stringsnap and (py-px)==d,
        touching = (py-px)==1,
        stringsnap = false

    action
        not stringsnap and (py-px)>d -> stringsnap = true
}

entity blockmover(blockL, blockR)
{
    action
        drivingL[blockL] and not stringtaut -> px = |px|-1,     [<]~
        drivingL[blockL] and stringtaut -> py = px+d; px = |px|-1,[<]--
        drivingR[blockR] and not stringtaut -> py = |py|+1,    ~[>]
        drivingR[blockR] and stringtaut -> px = py-d; py = |py|+1,--[>]
        drivingR[blockL] and not touching -> px = |px|+1,     [>]..
        drivingR[blockL] and touching -> py = px+1; px = |px|+1,[>][?]
        drivingL[blockR] and not touching -> py = |py|-1,      ..[<]
        drivingL[blockR] and touching -> px = py-1; py = |py|-1[?][<]
}


blockstate(); blockmover(x,y); handler(x); handler(y)
```

Conditions that give rise to interference and anomalous behaviour

[<]----[>]String under tension: conflict to be resolved
[>][<] Agents pushing against each other: conflict to be resolved
[<]----[<] Conflicts unless the agents cooperate ([>]----[>] , [<][<] , [>][>] are similar)
[<]~~[>]String can snap - and will under this model - if py-px = d-1
[>]....[<]Blocks collide if py-px = 2
[<]~~[<]Never generates interference ([>]~~[>] is similar)